SETLabs Briefings
VOL 9 NO 4
2011

Modeling and Programming with State Variables

By Vamsi Krishna B and Venkatesh Choppella PhD

Adopt state variables and fill the gaps left unattended in software composition by aspects and reactive programming languages

Software that stores state, responds to events and ensures that certain invariants are satisfied, is called reactive software. User interfaces and device drivers are examples of reactive systems. Wiring of various components yields a specific behavior to the system. Reactive programming languages like Flapjax allow composition on data invariants like y = f (x) to be specified during declaration. Similarly, aspects allow us to define composition in the form of point cuts. An example of a point cut is an entry point before a method call. However, both aspects and reactive programming languages do not target software composition using state. We extract the essence of aspects, which is deciding the control flow and add the declarative nature of reactive languages to allow software composition. We present state variables as the first step towards declarative composition. State variables are a variant of observer pattern, where the observers or handlers not only subscribe to a given event but also have the flexibility to add their own predicates representing the event or condition of their choice. We explore the possibilities of using state variables in application programming, device drivers and to extend implementations of observer pattern.

REACTIVE PROGRAMMING

Flapjax is a programming language built for ajax based applications that helps programmers write reactive programs for the web [1]. The primary motivation of this work is driven by the fact that callback mechanisms from the webpages are not properly typed to handle consistency of the system. Since Ajax applications make asynchronous calls to web services or the user interface elements to update of inform about change, the language promises to allow these declarations explicitly.

Thus elements or variables can be bound to a combination of other elements, variables or data from a URL. In this language or events from components of the system are stored as streams. For each component of the system a directed acyclic graph is built and maintained by the system to manage and track the relationships between components. Flapjax targets data dependencies and allows us to specify them in declaration. State variables, on the other hand, allow the control flow to be declarative which otherwise is implicit in the program.

Functional Reactive Programming or FRP is a framework for developing hybrid systems in a declarative style [2]. The framework introduces two polymorphic types namely *behavior* and *event*. A behavior is a data type that changes continuously over time. An event is a time ordered sequence of event occurrences. The implementation is called fran, built over Haskell. These two types are implemented using streams in Haskell. The goal of the language is to allow the programmer to extend the traditional call back model to support time. Time is a special behavior in the language. The state variable model also deals with extending callback. Here callbacks are extended not for temporal support but for custom predicates.

In the paper titled, "The foundations of Esterel" the author categorizes applications to be interactive or reactive and programming languages to be data oriented or control oriented [3]. The author introduces Esterel as a synchronous programming language developed to program reactive systems. It derives ideas from LUSTRE, SIGNAL and the State Chart formalism. Esterel differentiates between local variables and shareable signals. The statements of the language are classified as imperative and temporal. Describing design, semantics and implementation of the language, Berry et al., provide the language definitions and mathematical equivalents [4]. The language features support for patterns such as sequential, parallel, observer (watchdog). Statements such as emit, present, watching are unique in the language. The special constructs in Esterel, such as present and emit are treated as internal events of a system. State variables do not provide direct support for patterns such as sequential, parallel. A specific implementation of state variables is extended to support patterns, which is currently not published.

STATE CHARTS

State charts provide a visual means of modeling a system [5]. State charts operate by specifying a set of possible states for a system and defining the rules for transition. UML state diagrams, inspired by the original state charts, provide support for modeling dynamic behavior using states to complement class and sequence diagrams [5]. Most of the open and commercial UML tools provide means to convert the graphical elements to source code in a language of choice. However, not every implementation detail can be covered during the design phase using UML. Such an attempt will result in large unmanageable graphical forms. When an entity is programmed as an implementation of a state chart, it verifies the context and set of preconditions to be true and then evaluates an action assigned to the state. The key factor in designing a state chart is that the state space and the events generated and/or consumed at a given state are static. Thus traditional state charts or UML state diagrams are not quite suited for building compositions using state.

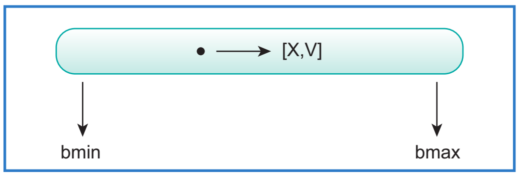

Figure 1: *Point in a Box*
Source: *Internal Research*

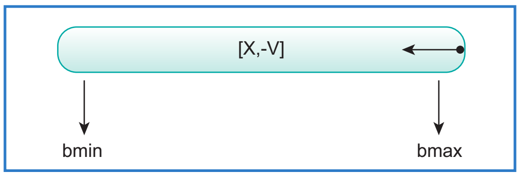

Figure 2: *Point in a Box : Boundary*
Source: *Internal Research*

OBSERVER PATTERN

Observer pattern is one of the popular design patterns proposed for software design [6]. The idea is to allow a set of observers to get notified when an event occurs in an observable. However most of the implementations of observer pattern, the regular event callback, delegates, signals-slots assume that the events generated by the observable are all implemented in it. In general the state of an observable may mean different to each observer even though the fields in the former look the same. Thus state variables have conditions or events installed from the observer. The state variable model tries to separate the event space both from the observer and observable, allowing its composition based on state.

ASPECT ORIENTED PROGRAMMING

Aspects are a way of separating the core functionality from the rest of the system [7]. Aspects define point cuts with in the program where a code segment can plug-in and get executed when the program control reaches the point cut. A point cut could be before executing display function, where an interested party will get notified before the function call to display happens, thus achieving a level of control on how program executes. State variables, on the other hand, can describe control flow based on state and uses internal events for communication. Thus state variables operate at a higher abstraction level.

DESCRIPTION

We will now go through an example that is small enough to start explaining the problem and our approach to the problem. The goal is to simulate the behavior of a particle in a box [Fig.1]. The particle has a position x and velocity. The box has boundaries in a given direction. The particle moves in the box and the only constraint being (Fig. 2) the particle has to stay within the box. In other words, the direction of particle is determined by the current position or state of the particle. We assume that the particle moves only in x direction. Let us write F as shown in Figure 3. The pseudo code describes behavior of the particle. It updates the value of the x by adding the distance it travels according to the velocity. In addition it reverses the velocity if it's close to the boundary.

PROBLEM DEFINITION

Change in behavior of a system with state is handled by conditional statements across the system. In case of the example described earlier, the change in x demanded a check on boundary condition using F.

```
F([x,v],u) =
        if (! crossingBoundary?([x,v]))
        then
            return [X+V*dt,v]
        else
        return [X+V*dt,-V]
```

Figure 3: *Compute Next State*
Source: *Internal Research*

The next state function F does not propagate the changes it makes to x or v instead it returns the changes made. If the invariants such as boundary condition are taken care by adding a conditional statement the system would not be declarative. The combination of a conditional followed by business logic can be split into a condition and a handler. Conditions are treated as internal events and handlers as their callback functions. The report focuses on how systems can be designed and programmed so that one can specify the progress of a system in a declarative way using custom events.

APPROACH

In the example as depicted in Figure 1, the two variables that change are x and v. Let us define a generic variable s with the representation shown in Figure 4. This image alone is the crux of the process that we describe to build software. Reading from S is similar to reading any variable. Writing to S, i.e., set(), not only writes to the store but also generates and sends out a pair of values representing the previous value and the new value of the variable. Let us call it O. The signal O acts as the catalyst to drive several other components. A system has several such variables and a set of invariants to satisfy. We also define two other components condition and handler that are responsible for creating the communication among the variables in a system so that the invariants are satisfied. Invariants could be as simple as x is always greater than 0 or might involve a predicate combining several variables that should be true. A condition is a carrier for a predicate and set of handlers interested in the predicate.

A handler is any executable unit that carries the business logic to be executed once a signal comes from the corresponding condition. Given representation of a state variable in the Figure 4 we continue to model our original example of particle in a box. To recall, the only constraint of the example was that the particle has to remain within the box. Also we saw that the velocity of the particle determines the next move. At each step we look for the boundary and we reverse the velocity if true. We now model the same using state variables as the basic components. The circuit representation of the same is presented in Figure 5. In this example x and v determine the key properties of the particle and are thus chosen as state variables. Let us assume b_{max} and b_{min} to be the extremes of the box. The boundary predicate is true when $x >= b_{max}$ and v is positive (or) $x <= b_{min}$ and v is negative.

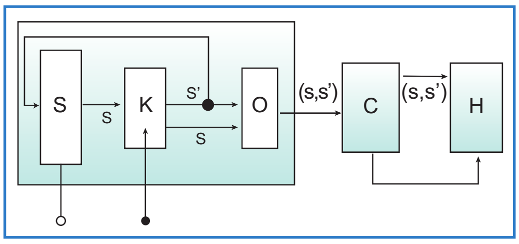

Figure 4: *State Variable Circuit*
Source: *Internal Research*

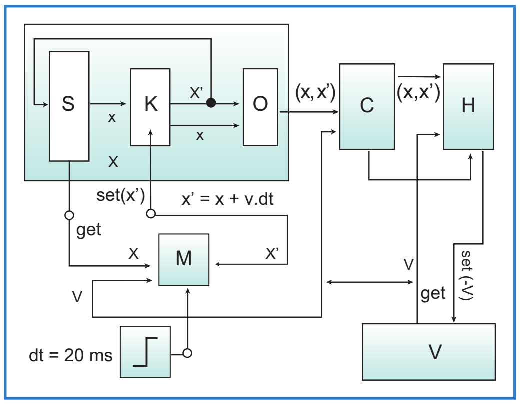

Figure 5: *Particle in a Box Circuit*
Source: *Internal Research*

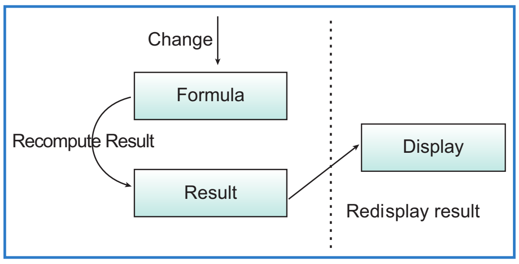

Figure 6: *Cell in a Spreadsheet*
Source: *Internal Research*

This computation is encapsulated in the component C. If the condition is true we should set V as -V, which is the job of H. M is the central component that represents the entire system. On each clock edge, the current x and v are read and next x i.e., x′ is computed as x = x + v.dt, with unit time and unit velocity. If we assume x starting at bmin and velocity as v = +1, each clock will increment x. Upon each write of x the boundary condition is checked. If the predicate is true, as described previously, the handler H is triggered which reverses the velocity.

APPLICATIONS

We present two examples from different domains which demonstrate programming with state variables.

Cell in a Spreadsheet

We model the behavior of a cell in a spreadsheet using two approaches. A cell contains primarily two fields, a formula and a result [Fig. 6]. The invariant of the cell is Result = eval(Formula). One way to achieve the same is to recomputed result each time the formula is changed. Figure 7 demonstrates one possible implementation.

Using state variables, to change the formula we only set its value [Fig. 8]. The dependents on formula will register with it during initialization on existing or custom [Fig. 9].

```
Cell: setFormula[formula] -> void
begin
    cellformula = formula
    result = eval (formula)
end
```

Figure 7: *Cell Implementation*
Source: *Internal Research*

```
Cell: setFormula [formula] -> void
begin
    cellformula.setValue(formula)
end
```

Figure 8: *Cell with State Variable*
Source: *Internal Research*

```
Cell: wireComponents[]-> void
begin
    cellformula. addHandler (“change”, Cell:
    recomputeResult)
end
```

Figure 9: *Wiring Handler*
Source: *Internal Research*

```
Cell: recomputeResult [oldformula, newformula]
        ->void begin
        // other verifications if needed
         cellresult.setValue(eval(formula))
end
```

Figure 10: *Handler to Recompute*
Source: *Internal Research*

Our interest is to change the value of the formula, we change it. A special initialization subscribes interested routines, recompute result, to necessary state variables [Fig. 10].

Device Driver

Tic Tac Toe is a well known two player game. The game is played on a grid of size nxn [Fig. 11]. At each move a player owns a location.

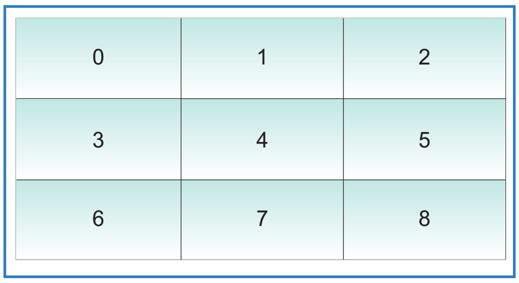

Figure 11: *Tic Tac Toe Grid*
Source: *Internal Research*

If a player owns any one of the valid combinations, the player is declared as the winner. Device driver is a program logic that is responsible for establishing communication between hardware and the processes in an operating system. A game is about progress of state, so is a device. We attempt to combine device drivers as a domain and the game as a requirement. The dynamics in the game is to determine if there is a winner.

There are 8 cases of completion—{0,1,2}, {3,4,5}, {6,7,8}, {0,3,6}, {1,4,7}, {2,5,8}, {0,4,8} and {2,4,6}. Each case is a tuple of three locations. If the tuple is filled with the same value then the tuple is complete and thus there is a winner. Figure 12 shows one possible combination that completes the game. Let us look at how we compute the same. We present the case in two approaches.

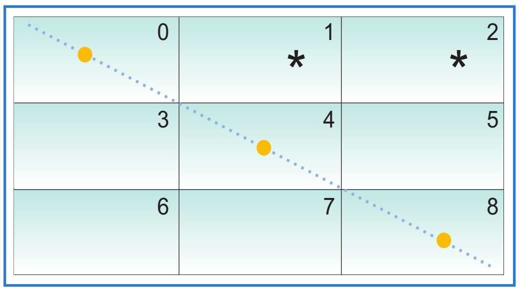

Figure 12: *Tic Tac Toe - Completion*
Source: *Internal Research*

A brute force method is used to compute the state of the game in both the cases. The first approach demonstrates a conventional way to calculate the game state after each successful write. Figures 13 and 14 present the implementation for the conventional approach.

```
write [turn, pos] -> boolean
  begin
     if checkturn (turn) and checkmove
(pos)
        do
           grid [pos] = turn
           computestate ()
           chanegturn ()
        end
     end if
  end
```

Figure 13: *Tic Tac Toe - Traditional Implementation*
Source: *Internal Research*

```
computestate[] -> boolean
begin
foreach (case in cases)
  do
   // validate case
   if(case is valid)
   do
       concludegame()
       return
        end
     end if
 end
```

Figure 14: *Tic Tac Toe - Compute State*
Source: *Internal Research*

```
write[turn, pos] -> boolean
begin
   if checkturn (turn) and checkmove
 (pos)
   do
      set(grid, pos, turn)
      changeturn()
      end
   endif
end
```

Figure 15: *Tic Tac Toe using State Variables*
Source: *Internal Research*

```
computestate [] -> void
 begin
  foreach (mask in masks)
  do
     addCondition (grid, Condition (mask))
     addHandler (grid, mask, declarewinner)
 end
end
```

Figure 16: *Tic Tac Toe: Wiring Masks to Grid*
Source: *Internal Research*

```
mask1.predicate [grid] -> boolean
begin
   if ( grid[0] == turn & gird [1] == turn & grid
   [2] == turn)
      return true
   endif
end
```

Figure 17: *Tic Tac Toe: Mask*
Source: *Internal Research*

```
declare winner [winner] -> void
   begin
      set (grid, 11, winner)
   end
```

Figure 18: *Tic Tac Toe: Declare Winner*
Source: *Internal Research*

The other approach is to separate the action of writing to the device or making a move from the action of recomputing state. In this case let us consider the grid to be a state variable. We use get(grid,position) and set(grid,position) to read and write values to the grid. We model each case of interest (called a mask) to be a condition. Figure 17 demonstrates an implementation for a mask or a condition. In a grid if positions 0,1,2 are marked by the same player then we have a winner. We add 8 conditions (mask1 to mask8) and the same handler (Fig. 18) to each one of them.

CONCLUSION

State of a software component or a data structure evolves over time and usage. Every change results in a set of events and other components respond to the events. Reactive languages like Flapjax, Esterel make the dependencies between data components implicit through declaration. State charts on the other hand make transitions of state implicit by definition in the model. Aspects provide a control of program execution by allowing arbitrary source to be executed at predefined set of points in the source code. State variables on the other hand makes control flow declarative and explicit overstate. Thus software compositions are done through custom predicates on the state of a system.

Refinement over Observer Pattern

Several implementations of observer pattern exist today. The event space is decided by the observable and not the observer. When the observable evolves each observer may want to look at the former in their own view. It is very certain that the observers verify the state of the observable after the notification. State variables will have the predicates or the conditionals being specified explicitly over the state. Thus the event space is moved out of observable and observer.

Software Composition

In software composition individual units depend on the state of each other in order to satisfy invariants of the entire system. Each component expects a set of specific events from a component of interest based on state. Thus the event space from a component is not static. Modeling a component as a state variable allows other components to register the events and build the composition around these events. Since the handlers are generic, they can be used to rewire the system based on state.

REFERENCES

1. Meyerovich, L. A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A. and Krishnamurthi, S. (2009), Flapjax: A Programming Language for Ajax Applications, ACM SIGPLAN Conference on Object-oriented Programming Systems.
2. Zand, W. and Hudak, P. (2000), Functional Reactive Programming from First Principles, in the Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pp. 242–252.
3. Berry, G. (1998), The Foundations of Esterel, Ecole des Mines de Paris.
4. Berry, G. and Gonthier, G. (1992), The Esterel Synchronous Programming Language: Design, Semantics, Implementation, Science of Computer Programming
5. Gogolla, M. and Presicce, F.P. (1998), State Diagrams in UML: A Formal Semantics using Graph Transformations, in the Proceedings of the ICSE'98 Workshop on Precise Semantics of Modeling Techniques, Technical Report TUM – I9803, pp. 55-72.
6. Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (2002), Design Patterns: Abstraction and Reuse of Object-oriented Design, in Broy, M. and Denert, E. (eds) Software Pioneers, Springer-Verlag New York, pp. 701-717.
7. Kiczales, G., Lamping, J., Mendhekar. A., Maeda, C., Lopes, C., Loingtier, J-M. and Irwin, J. (1997), Aspect-oriented Programming, ECOOP, pp. 220-242.